Wissensrepräsentation

Wintersemester 2014/15
HTW

Belegaufgabe

(Abgabe: 31. März 2015)

Eine Beispiel (siehe Anhang) zur Ermittlung von Verwandtschaftsbeziehungen ist in einem höherem Konzept als XML des Semantic-Web-Frameworks darzustellen und die Frage danach, welche Personen in der Beziehung der Base 2. Grades stehen durch ein automatisches Verfahren aus dem Gebiet des Semantic Web automatisch abzuleiten.

Hinweise:

- Das dargestellte Problem hat die Struktur eines zyklischen Netzes. Das Problem kann im Bedarfsfall (z.B. nicht behebbares Auftreten einer Endlosschleife beim automatischen Folgern) durch Löschen von Regeln auf ein nichtzyklisches Netz oder auch eine Baumstruktur reduziert werden.
- Die Regeln können (manuell; durch das beigefügte Prolog-Programm (Anhang 2); durch ein beliebiges Programm) in eine geeignete logische Form vortransformiert werden, so dass keine Regeltransformation im gewählten SW-Formalismus erforderlich ist.

Inhalt der Belegarbeit:

a) Beschreibung des Logischen Problems und von Lösungsansätzen in Logischer und Meta-Logischer Programmierung (KI-Ansatz)
b) Beschreibung eines gewählten Lösungsansatzes im Semantic-Web-Formalismus (SW-Ansatz)
c) Erfahrungen bei der Realisierung im Semantic-Web-Formalismus und Beschreibung von auftretenden Problemen und Beschreibung der Problembehandlung
d) Diskussion von Testläufen und -ergebnissen
e) Vergleich von KI- und SW-Ansatz (z.B. Programmierung, Übersichtlichkeit der Darstellung, Transparenz der Abarbeitung, Effizienz, ...)
f) Quellcode beschriebener SW-Programmier-Ansätze
g) Résumé und Ausblick

Anhang 1 - Beispielproblem

weiblich(sibylla).

weiblich(herrad).

weiblich(humilitas).

```
maennlich(wiligis).
maennlich(grigorss).
bruder(sibylla, wiligis).
schwester(wiligis, sibylla).
sohn(sibylla, grigorss).
sohn(wiligis, grigorss).
tochter(sibylla, herrad).
tochter(grigorss, herrad).
tochter(sibylla, humilitas).
tochter(grigorss, humilitas).
ehemann(sibylla,grigorss).

/*BASE1: Die Schwester eines Elternteils     */
  elternteil(X,Z) & schwester(Z, Y) => base1(X,Y).
/*BASE2: Die Tochter von Onkel oder Tante. .*/
  (onkel(X,Z) '|' tante(X,Z)) & tochter(Z,Y)  => base2(X,Y).
/*VETTER: Der Sohn von Onkel oder Tante.     */
  (onkel(X,Z) '|' tante(X,Z)) & sohn(Z,Y)  => vetter(X,Y).
/*COUSINE: Synonym zu Base2.                 */
   base2(X,Y) => cousine(X,Y).
/*COUSIN: Synonym zu Vetter.                 */
  vetter(X,Y) => cousin(X,Y).
/*ONKEL: Der Bruder des Vaters oder der Mutter.  */
  (vater(X,Z)  |  mutter(X,Z)) & bruder(Z,Y) => onkel(X,Y).
/*TANTE: Die Schwester des Vaters oder der Mutter. */
  (vater(X,Z)  |  mutter(X,Z)) & schwester(Z,Y) => tante(X,Y).
/*ELTERN: Vater und Mutter                   */
  vater(X,Y) &mutter(X, Z) => eltern(X,[Y,Z]).
```

```
/*ELTERNTEIL: Vater oder Mutter..                    */
   (vater(X,Z) | mutter(X,Z))  => elternteil(X,Z).
/*BRUDER: Männl. Kind des gemeinsamen Elternteils.          */
  elternteil(X,Z)& sohn(Z,Y) & Y\=X => bruder(X,Y).
/*SCHWESTER: Weibl. Kind  des gemeinsamen Elternteils.      */
  elternteil(Y, Z) & tochter(Z, X) & Y \= X  => schwester(Y, X).
/*NEFFE: Der Sohn des Bruders oder der Schwester.           */
  (bruder(X,Z)  |  schwester(X,Z))&sohn(Z,Y) => neffe(X,Y).
/*NICHTE: Die Tochter des Bruders oder der Schwester.       */
  (bruder(X,Z)  |  schwester(X,Z))&tochter(Z,Y) => nichte(X,Y).
/*GESCHWISTER: Bruder oder Schwester                        */
  (bruder(X, Y) |   schwester(X,Y)) => geschwister(X, Y).

/*VATER; Elternteil ist männlich                            */
   elternteil(X, Y) & maennlich(Y) => vater(X,Y).
/*MUTTER; Elternteil und weiblich                           */
   elternteil(X, Y) & weiblich(Y) => mutter(X,Y).
/*VATER/MUTTER-Beziehung zur Tochter */
   tochter(X, Y) => vater(Y,X)&maennlich(X) | mutter(Y,X)& weiblich(X).
/*VATER/MUTTER-Beziehung zum Sohn */
   sohn(X, Y) => vater(Y,X) &maennlich(X)| mutter(Y,X) & weiblich(X).
```

Anhang 2 – Logik-Programm (KI-Ansatz)

```
%%%% Verwandschaftsbeziehungen %%%%%%%%%%%%%%%%%%%%%%%%
%% Aufrufe:
%% ?- transall(L).                  % Zur Transformation der Regeln in Prolog-Klauseln
%% ?- tiefe(base2(X,Y), [], _).   % Zur Aufruf-Arbeitung (mit Schleifen-Test)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%Trace-Funktionalitaet durch Umdefinieren von writelnProt/1 und writeProt/1%%%

%Protokollierung aus

writelnProt(_).

writeProt(_).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


:- op(1150,xfx, '=>').

:- op(1000, xfy, '&').

%!!! Daten des Problembeispiels hier integrieren

/*** Meta-Programm zur Aufruf-Abarbeitung (mit Schleifentest)  ****************************/

tiefe(true,Erg,Erg):- %Programm unvollständig; aber BIP-Aufrufe werden behandelt

   writelnProt('ENDE'),!.

tiefe((P,Q),Acc,Erg) :- writelnProt(port(1-',')),        %Konjunktion

   !, tiefe(P,Acc,Zwi),

   tiefe(Q,Zwi,Erg).

tiefe((P ; Q),Acc,Erg) :-  writelnProt(port(2-':')),      %Disjunktion

   !, (tiefe(P,Acc,Erg);

   tiefe(Q,Acc,Erg)).

tiefe((not(P)),Acc,Acc) :-

   writelnProt(port(3-'not'-mit-not(P))),

   !, (tiefe(P,Acc,_)->(writelnProt('Port3':not(P)-nicht_erfolg),false)

                        ; (writelnProt('Port3':not(P)-erfolg),true)).

tiefe(P,Acc,Acc) :-                                        %Behandlung von built-in-Prozeduren

   writelnProt(port(4-'bip'-P)),

   bip(P),!,call(P).

tiefe(P,Acc,Erg) :-

   writelnProt(port(5-'cl')),writelnProt('Call'(P)-withAcc:Acc),
```

```
   not(member(P,Acc)),       %Aufruf P und Proz.kopf P
   %copy(P,Pfresh),
   Neu=[P|Acc],
   clause(P,K),                                    %Ermittl. des Proz.körpers K
   %bt(writelnProt(neueKlausel-P:-K)),
   %not(member(P,Acc)),
         %Neu=[P|Acc],
         %write('  '),
         %writeln('Clause':(P:-K)-'Accu:':Neu),
   tiefe(K,Neu,Erg).         %Abarbeitung des Proz.körpers K

bip(Head) :-
  predicate_property(Head,built_in).
/*****************************************************************************/
copy(P,Pfresh) :-
   asserta(dummy(P)), retract(dummy(Pfresh)).

/*** Transformationenen ************************************

A->B & C = (A->B)  & (A->C)
A->B|C  = (A&~C->B)   &  (A&~B->C)
A|B -> C = (A->C) & (B->C)
A&B -> C =  A&B -> C
A-> B  =  A->B
A = A
 ***************************************************************/
/*Einfaches Metaprogramm zur Transformation: A, B, C werden als unstrukturiert behandelt)*/
trans((A=>B&C)) :-
```

```
  !, trans((A=>B)), trans((A=>C)).
trans((A=>B|C)) :-
  !, trans((A & not(C)=>B)), trans((A & not(B)=>C)).
trans((A|B=>C)) :-
  !, trans((A=>C)), trans((B=>C)).
trans((A & B=>C)) :-
  !, transcond(B,Bn),
    assertz((C :- A, Bn)).
trans((A=>B)) :-
  !,  assertz((B :- A)).
%trans(A) :-
%    assertz(A).

% Meta-Programm zur Wissenstransformation
% Logische Folgerungsbeziehungen werden in Prolog-Prozeduren transformiert
% A, B, C sind ausschließlich (bis auf eine Ausnahme) positive Atomformeln
% In A&B => C kann B eine Konjunktion positiver Atomformeln sein
trans((A=>B&C),[L1,L2],SIM) :- !,
  trans((A=>B),L1,SIM), trans((A=>C),L2,SIM).
trans((A=>B|C),[L1,L2],SIM) :- !,
  trans((A&not(C)=>B),L1,SIM), trans((A&not(B)=>C),L2,SIM).
trans((A|B=>C),[L1,L2],SIM) :-
  !, trans((A=>C),L1,SIM), trans((B=>C),L2,SIM).
trans((A&B=>C),[C],SIM) :- !,
    transcond(A,A1),
    transcond(B,Bn),
    (SIM=0->assertz((C:-A1,Bn));writeln((C:-A1,Bn))).
trans((A=>B),[B],SIM) :- !,
```

```
    (SIM=0->assertz((B:- A));writeln((B:-A))).



%Fall: Konjunktion von Bedingungen
transcond((not((B1&B2))),(not(C1); not(C2))) :-
  !,transnot(B1,C1),
    transnot(B2,C2).
transcond((not((B1|B2))),(not(C1),not(C2))) :-
  !,transnot(B1,C1),
    transnot(B2,C2).
transcond((B|B1),(C;C1)) :-
  !, transcond(B,C),
     transcond(B1,C1).
transcond((B1&B2),(C1,C2)) :-
  !, transcond(B1,C1),
     transcond(B2,C2).
transcond(B,B).

transnot((B1&B2),(C1,C2)) :-
    transcond(B1,C1),
  !,transcond(B2,C2).
transnot((B1|B2),(C1;C2)) :-
  !, trancond(B1,C1),
     transcond(B2,C2).
transnot(B,B).

transrule(B=>F,L,SIM) :-
```

```
  (B => F) ,
  trans((B => F),L,SIM).
transrule(_,_,_).

transall(List) :-
  findall(F0, transrule(_,F0,1) , L0), writeln('L0':L0),
  flatten(L0,[],L1),writeln('L1':L1),
  list_to_set(L1,L),  %swi-Prolog
  writeln('Set':L),
  abolish_all(L),
  findall(F, transrule(_,F,0), List).

pp([]).
pp([A|B]) :-
  functor(A,F,N),
  (current_predicate(F/N)->print_procedure(F/N);true),
  pp(B).

print_procedure(F/N):-
  functor(F1,F,N),
  clause(F1,K),writeln((F1 :- K)),
  fail.
print_procedure(_).
abolish_all([]).
abolish_all([A|_B]) :-
  %functor(A,F,N),
  %abolish(F/N),
        clause(A,K), %functor(A,weiblich,_N)->writeln('Auswahl': A:-K);true),
```

```
    (K\=true->retract((A:-K));true),
     fail.
abolish_all([_A|B]) :-
  abolish_all(B).

bt(_).
bt(Call) :- call(Call),!,fail.

flatten([],Xs,Xs).
flatten([X|Xs],Ys, Zs) :-
    atom(X),%not(X=[]),
          !,flatten(Xs, [X|Ys], Zs).
flatten([X|Xs],Ys, Zs) :-
    not(X=[_|_]),
          !,flatten(Xs, [X|Ys], Zs).
flatten([X|Xs],Ys, Zs) :-
   flatten(X,Ys1, Zs),
   !,flatten(Xs, Ys, Ys1).
```